Causal Models in the Acquisition and Instruction of Programming Skills

Brian Reiser

Princeton University

Research and Advanced Concepts Office
Michael Drillings, Acting Director

January 1995

19950421 069

United States Army
Research Institute for the Behavioral and Social Sciences

U.S. ARMY RESEARCH INSTITUTE
FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency Under the Jurisdiction
of the Deputy Chief of Staff for Personnel

EDGAR M. JOHNSON
Director

Research accomplished under contract
for the Department of the Army

Princeton University

Technical review by

Joe Psotka

NOTICES

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 1995, January	3. REPORT TYPE AND DATES COVERED Final Sep 87 – Aug 90	
4. TITLE AND SUBTITLE Causal Models in the Acquisition and Instruction of Programming Skills			**5. FUNDING NUMBERS** MDA903-87-K-0652 01102A 20161102B74F TA 3901 WU C23
6. AUTHOR(S) Reiser, Brian (Princeton University)			

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Princeton University Department of Psychology Washington Road Princeton, NJ 08544	8. PERFORMING ORGANIZATION REPORT NUMBER --
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Institute for the Behavioral and Social Sciences ATTN: PERI-BR 5001 Eisenhower Avenue Alexandria, VA 22333-5600	10. SPONSORING/MONITORING AGENCY REPORT NUMBER ARI Research Note 95-15

11. SUPPLEMENTARY NOTES

Contracting Officer's Representative, Michael Drillings.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE --

13. ABSTRACT (Maximum 200 words)

     This research project investigates how an interactive learning environment can support students' learning and acquisition of mental models when acquiring a target cognitive skill. In this project, we have constructed GIL, an intelligent tutoring system for LISP programming, and have used GIL to conduct pedagogical experiments on skill acquisition. We have studied two ways in which an interactive learning environment can facilitate students' acquisition of novel complex domains. The first set of studies examines how explanatory feedback, generated from the system's problem-solving knowledge, can facilitate students' learning. The experiments demonstrate computer-based support during learning can help students construct a more effective model for reasoning in complex domains.

14. SUBJECT TERMS Skill acquisition Intelligent tutoring systems Mental models	15. NUMBER OF PAGES 41
	16. PRICE CODE --

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited .

Contents

1   Introduction and Overview

Individualized instruction is often thought to be the most effective form of instruction, particularly for problem solving domains (e.g., Bloom, 1984; Cohen, Kulik, & Kulik, 1982). The promise of modern educational computing is the opportunity to deliver the benefits of individualized instruction to students via computers. There are a number of different approaches embodied in current forms of computer instruction. For example, some have argued that the best use of computers in instruction is to provide a context in which students can explore a domain and learn by discovering its principles (Papert, 1980; Schank & Farrell, 1987; Schwartz, 1989). Others have argued that systems should provide more directive feedback during learning (e.g., Anderson, Boyle, & Reiser, 1985; Genesereth, 1982; Kimball, 1982). While there is no general agreement about which instructional model holds the most promise for computer-based instruction, the effectiveness of expert teachers in one-on-one tutoring settings suggests that it is a profitable methodology to emulate (Carbonell, 1970; Collins, Warnock, & Passafiume, 1975).

Is it possible to capture the instructional benefits of individualized tutoring in a computer system? This project investigates how an interactive learning environment can support students' learning and acquisition of mental models when acquiring a target cognitive skill. In particular, the research program explores how graphical representations and intelligent guidance can facilitate students' learning in a new domain.

Our approach is to construct an intelligent interactive learning environment, and to use the learning environment to conduct pedagogical experiments on skill acquisition. We have constructed an intelligent tutoring system called GIL (*Graphical Instruction in LISP*), which teaches students to program in LISP. GIL functions in both a guided tutoring mode and as a more open-ended exploratory learning environment. In its guided tutoring mode, GIL provides explanatory feedback in response to student errors or requests for hints by continually comparing the student's solution as he or she constructs it to the range of solutions suggested by its problem solver. In its exploratory mode, GIL provides support for students to articulate their hypotheses about how a program will behave, and tools for testing their hypotheses. In this report, we review our research on GIL, including the knowledge representations and graphical techniques employed, and the empirical studies of students' learning performed using these systems. Our results suggest that both the graphical representation and the causal explanations provided by the system account for GIL's pedagogical effectiveness. The graphical representations and causal explanations work together to help students build and capitalize on an effective mental model for programming.

We present results from two sets of studies that explore how GIL facilitates

students' learning:

- how a graphical representation facilitates reasoning about devices.
- how model tracing guidance helps students learn by repairing their own errors.

2 GIL: An Intelligent Tutoring System for Programming That Explains Its Reasoning

Human tutors carefully monitor students' problem solving and provide frequent feedback that can be very subtle (Merrill, Reiser, Ranney, & Trafton, 1991). Can this type of reasoning be modeled in a computer tutor? In this section, we discuss how GIL is designed to support students' problem solving.

What capabilities would be required to achieve some of the pedagogical benefits of a human tutor in an intelligent tutoring system? At a minimum, a system must be able to follow students' reasoning and detect when they have made an error or embarked on a bad strategy. The system should also be able to provide suggestions in these situations and give hints when students are stuck. Intelligent tutoring systems typically achieve these goals by using three distinct types of knowledge. First, the system should be able to solve the problems in the target area. Expert problem solving plans and rules are often used to represent this domain knowledge (Anderson, Boyle, & Reiser, 1985; Clancey, 1987). Second, the system needs to represent its view of the student's understanding of the domain. This representation is typically called the student model. The third type of knowledge implements pedagogical strategies.

In the *model tracing* methodology, the tutor follows students' solutions and identifies errors by matching the students' problem solving steps with the reasoning of an underlying rule-based domain expert (Anderson, Boyle, Corbett, & Lewis, 1990; Anderson, Boyle, Farrell, & Reiser, 1987; Anderson et al., 1985). This matching is used as the basis for providing ongoing feedback to students while they progress through a problem. The general strategy in model tracing systems is to present a problem for the student to solve, track the student's progress step by step, and intervene with explanatory feedback upon an error or a request for help. For example, an incorrect use of a programming concept might trigger a brief explanatory message associated with the buggy problem solving rule that captures the misconception. In this situation, the feedback helps the student diagnose the error and suggest a way to approach its repair. A model tracing system may also respond to an error by finding a correct rule embodying an appropriate action in the current problem solving context. In this case, the tutor guides the student toward a correct replacement step for the error. In contrast to these situations, if the student's step

is one that would be produced by executing one of the correct rules considered by the system, the tutor silently follows the student's path through the problem. The model tracing methodology has formed the basis for a number of intelligent tutors that teach computer programming (Anderson, Conrad, & Corbett, 1989; Anderson & Reiser, 1985; Reiser, Anderson, & Farrell, 1985), proof construction in geometry (Anderson, Boyle, & Yost, 1986), solving algebraic equations (Milson, Lewis, & Anderson, 1990), and calculus (Singley, 1990).

GIL (*Graphical Instruction in LISP*) is an intelligent tutoring system that supports novice students' problem solving as they learn computer programming in LISP (Reiser, Friedmann, Kimberg, & Ranney, 1988; Reiser, Kimberg, Lovett, & Ranney, 1991). The goal of our research on GIL is to investigate how to facilitate students' understanding of a new domain as they learn to solve problems. We are investigating two ways in which a tutor can facilitate reasoning. A first research goal is how to simplify problem solving in complex domains with a graphical representation that provides better scaffolding for students' reasoning than traditional (i.e., text) representations (Ranney & Reiser, 1989; Reiser, Friedmann, Gevins, Kimberg, Ranney, & Romero, 1988). A second research issue is how a tutor's explanations can guide students' reasoning and help them learn more effectively from their impasses, errors, and successes (Reiser, Friedmann, Kimberg, & Ranney, 1988; Reiser, Kimberg, Lovett, & Ranney, 1991).

2.1   Components of the GIL Tutor

The current version as of the end of this contract (8/90) is GIL 1.4. GIL is written in LOOPS and the Interlisp dialect of LISP and runs on Xerox LISP workstations. We are currently porting GIL to Common LISP on Macintosh computers. This port should be completed during the first year of the next contract (*Graphical Representations and Causal Models in Intelligent Interactive Learning Environments*, Contract MDA903-90-C-0123 to Princeton University).

The diagram in Figure 1 displays the modules that comprise GIL and the knowledge representations they use. GIL follows the model tracing methodology used in the tutors built by Anderson and his group (Anderson et al., 1985; Anderson et al., 1990; Reiser et al., 1985) but it contains a different type of problem solving model designed to make explicit the causal knowledge about programming operations, and an explanation component that constructs hints and error feedback directly from the content of its problem solving knowledge.

GIL understands each student step as it is generated. A step in GIL consists of selecting a LISP function and specifying the function's input and output. GIL consists of four main modules: a graphical interface, a response manager, a problem solver, and an explainer. The graphical interface is a graphical programming
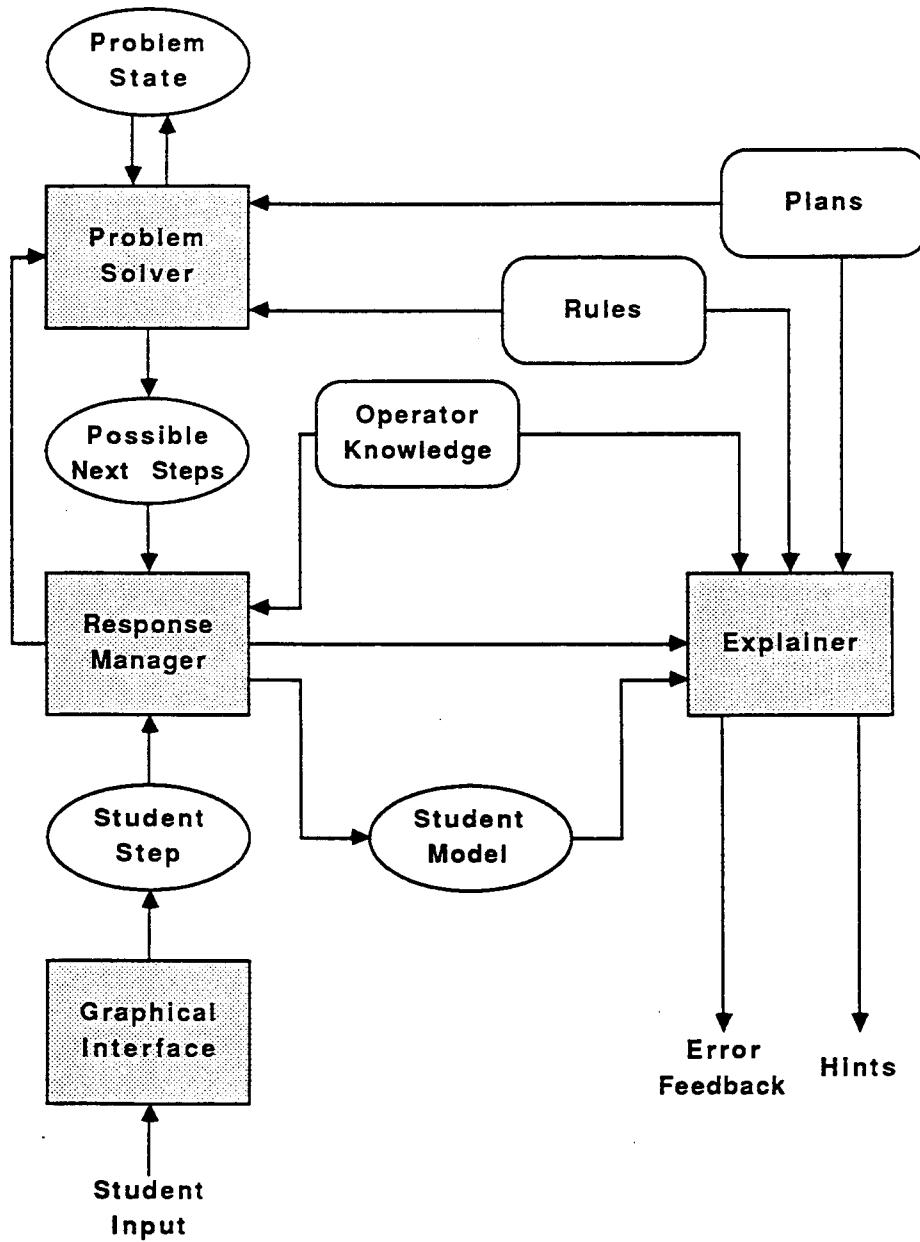
Figure 1: The architecture of the GIL programming tutor. The four shaded boxes represent GIL's four modules. The rounded boxes represent the knowledge bases used in model tracing, and the ovals represent data structures that GIL constructs as the student builds a solution.

environment in which students construct LISP programs. The response manager implements the model tracing process, drawing upon the problem solver and explainer. GIL processes each step taken by the student and determines whether it is on the path toward a solution or is an error. The analysis is performed by comparing the student step with the steps suggested by the problem solver. If an error is found, the explainer analyzes the discrepancies between the student's step and the closest matching correct rule and offers suggestions to the student about how to improve the step. Explanations may draw upon the problem solving rule, general knowledge about the operator being used, and the higher-level plan of which the step is a part.

2.2   The GIL Curriculum

The curriculum covered by GIL 1.4 is displayed in Table 1. The course is an introductory programming course, designed for novices. No prior programming experience is required to master the material in the curriculum.

The eight sections in the curriculum cover the material in the first three chapters of the textbook on which the curriculum is loosely based, *Essential LISP* by Anderson, Corbett, and Reiser (1987). In sections 1–2, students learn to write programs using a concrete example for each program. These programs construct and manipulate lists. In section 3, students learn to use variables, and to run their programs on new examples to test and debug them. Section 4 introduces arithmetic functions. Sections 5–8 cover conditional processing in programs. The concept of conditional processing is explained in section 5, and predicates for testing properties of data are introduced in section 6. Then, section 7 presents the LISP conditional structure, *cond*. Finally, logical functions are introduced in section 8.

At the completion of these 8 sections, students have learned how to create programs, test them on particular examples, and run them on new data. They have learned how to write programs that manipulate lists, and programs that behave conditionally depending on the characteristics of the input. They have learned a number of important programming techniques, including how to order the tests for a set of possibilities that are not mutually exclusive, how to use logical functions to specify complex contingencies, and how to design programs that exhaustively cover the set of possibilities.

In total, students have learned to use 24 LISP functions: 7 functions for manipulating lists (*cons, append, list, first, rest, last, reverse*), 4 arithmetic functions (+, -, *, /), 9 predicates (*listp, atom, null, numberp, equal, member, zerop*, <, >), 3 logical functions (*and, or, not*), and the conditional construct (*cond*).

The following two sections describe the principles employed in the design of GIL's graphical representation and in its model tracing guidance.

Table 1: The Curriculum Implemented in GIL 1.4

2.3 GIL's Graphical Programming Interface

GIL's graphical programming interface is designed to be congruent with the reasoning required in programming tasks. GIL students build a program by connecting icons representing program constructs in a graph, rather than by defining LISP functions in the traditional text form. In the early GIL curriculum, GIL students work out their algorithms by reasoning about the behavior of the program on a particular example. GIL students take a step by selecting a LISP function and specifying its input and output data, thereby making predictions about the changing state of the program's data. Figure 2 shows a partially completed solution to one of the more difficult list manipulation problems in the first two lessons of the GIL curriculum.

A first advantage of the graphical interface is that it provides a representation that makes the behavior of the program more visible. Students include each of the intermediate results in the program by specifying how the output for one function becomes the input for another. Thus, when a program in GIL is complete, it specifies how a chain of LISP functions transforms the original input data to achieve some particular target output. The internal states of the program — the data computed by the program on the steps between the initial input and final output — are invisible in the traditional text representation. In GIL's graphical representation, they are rendered explicit. Reasoning about dynamic objects such as the motion of physical objects, electricity in a circuit, or the behavior of machines and computer programs requires being able to propagate causes and effects to predict the states that occur between the initial and final states (e.g., Bayman & Mayer, 1988; de Kleer & Brown, 1983; White & Frederiksen, 1990; Young, 1981). Helping students understand and reason about these internal states can be crucial for helping novices learn to construct computer programs (Bayman & Mayer, 1988; du Boulay, O'Shea, & Monk, 1981).

A related second advantage of GIL's graphical interface is that it leads students to make their own reasoning explicit. By providing the intermediate results the program will compute, students articulate their predictions about how the program will behave while they are constructing it. In contrast, when students construct a text program they need only specify a sequence of operations, without articulating the input and result of each operation. GIL provides a natural medium for students to articulate their reasoning while progressing through a problem. Focusing students on the reasoning processes involved in solving problems rather than on the results is an important component of effective instruction (Brown, 1985; Collins & Stevens, 1982; Collins, Brown, & Newman, 1989). Such a system may also facilitate the students' own monitoring of their problem solving progress. Furthermore, the tutor can better provide feedback when students indicate what they expect will occur as the program runs. The GIL representation can be contrasted with the standard text representation in which only the final results of the students' reasoning are evident.
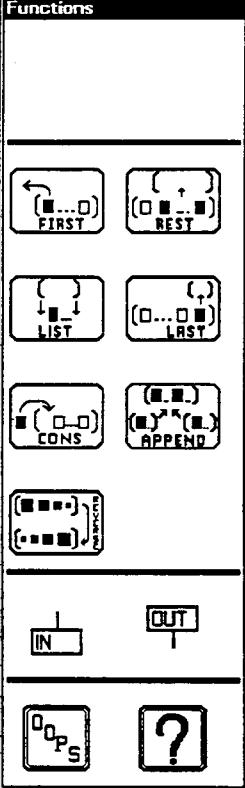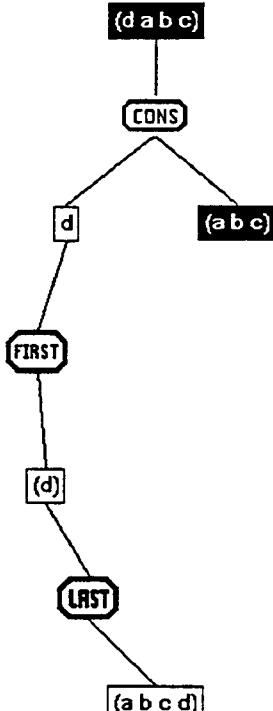
Figure 2: Solving a problem in GIL. The student has requested a hint about how to proceed.

Figure 3 shows a completed solution in GIL and the corresponding text form of this program.

A third advantage of the graphical representation is that it provides a closer fit with the structure of the student's reasoning. For example, reasoning chains in a GIL problem solution are represented by branches of the graph that converge to eventually achieve the final goal. Thus, translating a student's plan into components to be added to a GIL solution is simplified, and the plan's structure is preserved by the graph. In addition, the structure itself is more readily discernible in the graph than it is in the text representation, in which it must be extracted from more subtle cues such as indentation and number of parentheses. Graphical representations often facilitate inferences about the structure of a problem that are difficult with verbal descriptions (Fuson & Willis, 1989; Larkin, 1989; Larkin & Simon, 1987). Thus, students can more easily keep track of the current status of their solution plan and more easily monitor their solution progress. Furthermore, the closer correspondence of the graph with the structure of the student's reasoning allows students to focus less on syntactic details and more on the semantics of the constructs, and on combining operations to construct a plan.

Finally, the interface allows students to focus problem solving on individual solution components. Students can construct larger structures by first building and testing simple components. Students can break a problem into subgoals, such as two branches of a graph or different conditional cases, and then focus on one subgoal, obtaining feedback about the behavior of that portion of the graph. In addition, GIL allows students to make reasoning steps in whatever order they find natural, even though the order of these steps may not match the order in which the components appear in the final surface form of the solution (Reiser, Ranney, Lovett, & Kimberg, 1989; Trafton & Reiser, 1991).

One potential danger of the graphical representation is its potential limitations for representing complex problems. A tutor that uses a graphical representation can require a large workspace even for simple problems; as problems become more complex, such a detailed representation might overwhelm students and lead them to focus more than necessary on small aspects of the solution. One way to counter this problem is to modify the representation as the curriculum progresses. In the early lessons with GIL, for example, students are required to enter the intermediate products for each function they use. While helpful for beginners, more advanced students may find this distracting. Also, more complex problems would be unwieldy to write and debug since they would require so much workspace. As we shall see, in the later part of the GIL curriculum, students can skip these intermediate products. Similarly, when students move on to iterative functions the representation for conditional expressions becomes more concise.[1] Thus, the constructs learned in

---

[1]The additions to the GIL interface for the iteration lesson are now designed, and construction
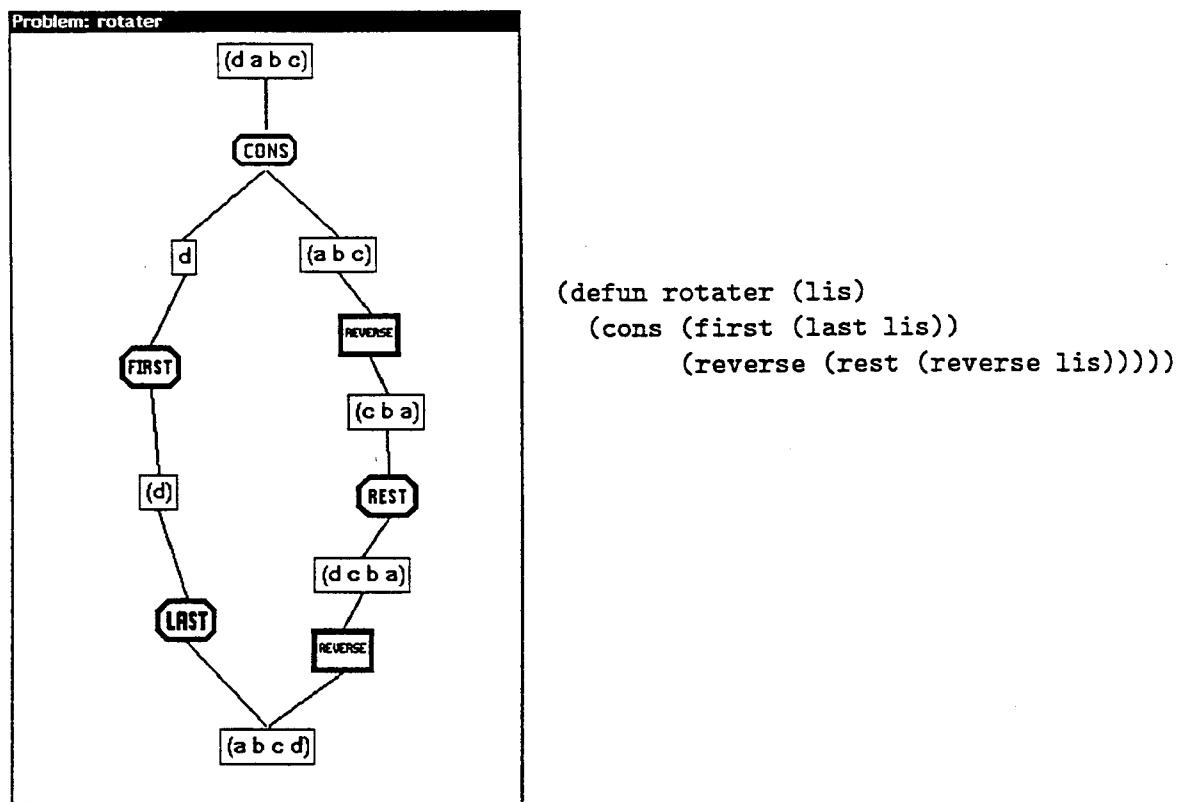
Figure 3: A completed program graph and the corresponding LISP code.

earlier lessons become more compact as students move on to new topics to allow more complex programs to be written.

2.4 Providing Guidance and Support for Students' Problem Solving

GIL's explanatory feedback and guidance is also designed to help students learn more effectively as they solve problems. GIL employs the model tracing technique to follow students' reasoning, offer assistance when requested, and determine feedback upon errors. GIL can follow any correct solution to the assigned problems. Most problems have many different possible solutions. For example, the problems in section 2 contain up to 14 unique solution graphs each, and up to 1720 different paths or different sequences of steps that a student can take in each problem.

GIL's problem solver contains knowledge about the behavior of the various LISP programming constructs, including how each LISP operator transforms data and what each accomplishes. In addition, the problem solver contains strategic knowledge, encoded as plans, enabling it to reason about how to assemble programming constructs to accomplish desired goals. Thus, the system can not only construct a sequence of LISP operators, but can also convey why each component is useful in a given situation. Figure 4 shows GIL's response to a *legal error* in the early curriculum; the step in the program does not correctly manipulate the selected data. In these situations, GIL explains how the student's step differs from LISP's actual behavior in that situation. In addition, the feedback contains strategic information — GIL suggests modifications of a step that will not only make the step legal, but also help in solving the current problem. GIL also responds to a second type of error called a *strategic error*, in which the chosen step is a legal LISP operation but does not appear to be part of any plan that correctly satisfies the problem constraints. GIL's feedback on strategic errors suggests modifications in the solution plan. An example of GIL's response to a strategic error is shown in Figure 5.

GIL provides two levels of help for most errors. The first level points out the nature of the error. The second level, provided only if requested by the student, offers a specific suggestion and explains why this suggestion would be effective. In this way, the first level of feedback provides an opportunity for students to figure out how to fix the step or to request more directive feedback (by asking for a further hint).

The degree of GIL's support varies with the student's progress through the curriculum. In early lessons, the focus of instruction is on learning the basic functions for manipulating lists. The student specifies a step in a solution by selecting a function and specifying its input and output data, using the example original input

---

should begin on this lesson in the first year of the next contract.

Problem: getends | **Assignment**

Write a program that takes one list as an argument and produces as output a list containing the first and last element of that argument. For example, if the argument were **(a b c d)** then the output would be **(a d)**.

Hints

Using **LAST** on **(a b c d)** to construct something is a good idea. However, d is not what **LAST** will produce. The correct output should be a list.

(d) is a list containing the last element of **(a b c d)**. Try using **(d)** as the output.

New Output   New Step

(a d)

a   d

FIRST   LAST

(a b c d)
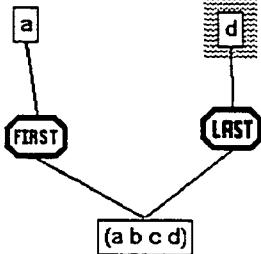
Figure 4: An explanation constructed by GIL in response to a legal error. Two levels of help are shown.
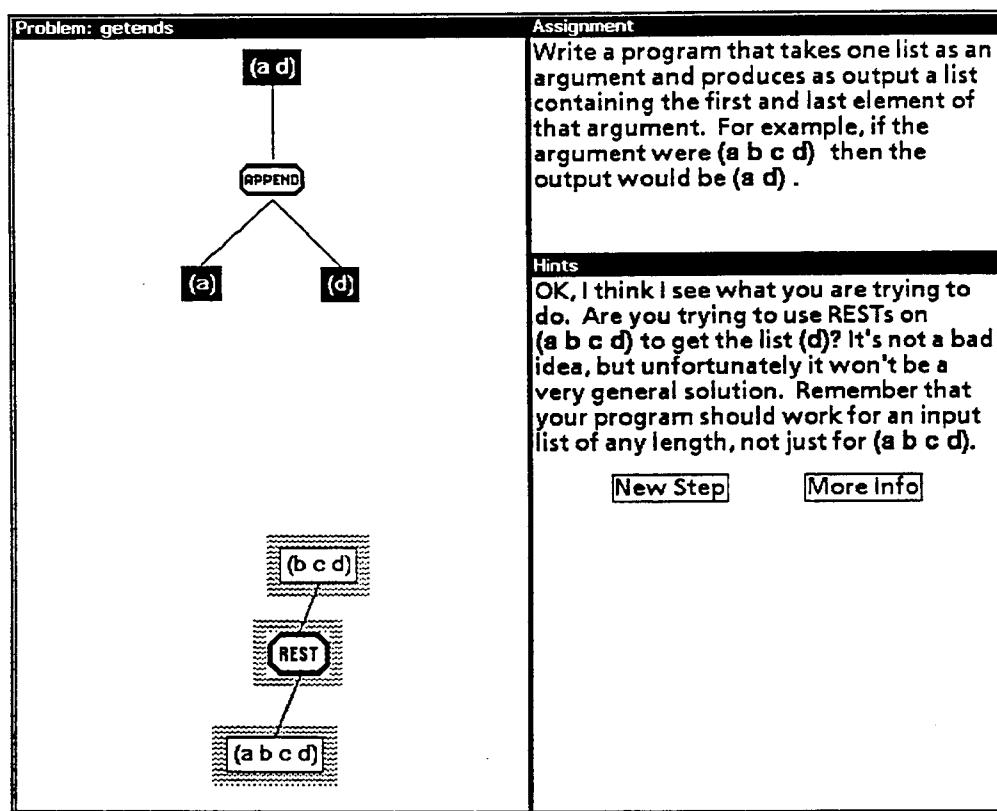
Figure 5: An explanation constructed by GIL in response to a strategic error.

and target output provided for the program (see Figures 2–5). These intermediate products help students work out their algorithms. In addition, these intermediate results provide an example around which GIL can structure its feedback.

In later lessons, the focus is on learning how to use predicates and logical functions to test properties of data, performing actions contingent upon these tests. In these later lessons, students see intermediate results on a more global level — the results of this conditional processing rather than of the list manipulation occurring within a given test or action. At this point in the curriculum, students are not required to supply the intermediate products of list manipulation (although they are available from GIL on request), because these subskills should be well learned from the previous chapters. Accordingly, GIL's model tracing feedback begins to concentrate on a more strategic level. GIL provides feedback when students misorder components of the conditional part of a program or when it becomes apparent they are pursuing inappropriate solution plans, such as an erroneous implementation of a desired test.

2.5   Providing Support for Students' Exploration

GIL also contains facilities for students to test their programs to determine whether they behave as expected. Using these facilities, rather than solely on GIL's model tracing feedback, students can take more of the responsibility for determining whether the program satisfies the problem constraints and, if not, which components are at fault.

Students can work with GIL in two different pedagogical modes. In one mode, GIL performs model tracing, intervening to point out errors or offer hints as students solve problems. The testing facilities GIL contains also makes it possible for students to learn successfully without model tracing. In this exploratory mode, GIL provides the graphical programming interface and testing facilities but leaves students free to explore and construct and repair their solutions without GIL's active intervention. When students work with GIL in this exploratory mode, they must find and repair their mistakes without GIL's model tracing guidance. Thus, students are allowed to explore, and can test their programs themselves when they want feedback.

GIL's testing facilities enable students to test any portion of a partially complete program. When students test a portion of a program, GIL reports whether the intermediate products in that part of the graph are correct. An example of a student learning in the exploratory version of GIL is shown in Figure 6. Here, the student tested the branch of the graph ending with the data *(b)*. Notice that the error detected by GIL in the test was committed in a part of the program the student had completed earlier. Notice also that in contrast to GIL's model tracing response shown in Figures 4 and 5, the exploratory environment merely points out how LISP's
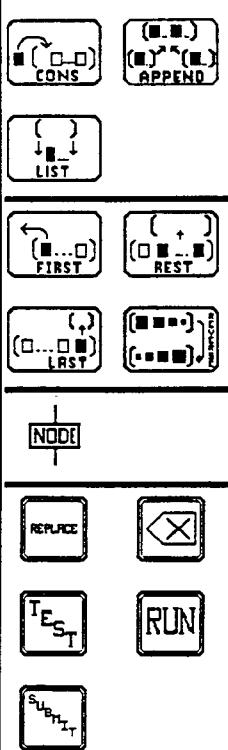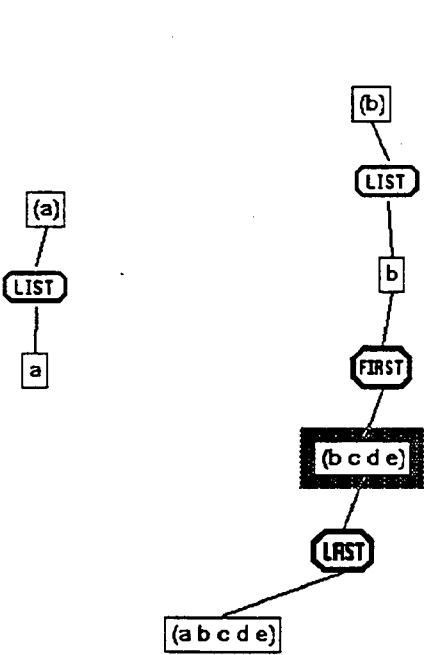
Figure 6: Testing a partial program in the exploratory version of GIL

behavior would be different from the student's program but does not explain the most strategic way to fix the error for this problem (which in this case would be to retain the output but replace the function).

Students can also test their programs, when complete, by running the program on new data, replacing the original input data. In this situation, the intermediate products and resulting output based on the new data are calculated by the system. Each node in the graph blinks as it is processed in the order that LISP would evaluate the expression. As it reaches each data node, GIL enters the value of the intermediate result that the function would return. Figure 7 shows an example of a student in the process of running the completed program on new data.

GIL can also enable students to explore freely on tasks of their own choosing, rather than working on assigned problems. Students can construct, test, and run whatever program graphs they wish. When students have constructed something of interest, they can then print the result and clear the screen to begin a new graph. Figure 8 displays a student saving the results of some exploration of the LISP functions she had been studying.

In the later part of the curriculum (sections 3–8), students use variables in their programs instead of writing a program with a particular example, and can run their programs at any point in their construction, providing input data for each variable. A button is provided in the menu for each variable the problem will require. The first time the student selects each of the variable buttons, he or she will be prompted to provide a name for that variable, given a description of its role in the problem (e.g., "the target item," "the list being searched").

The conditionals curriculum (sections 5–8) is a major step forward in the power of the programs students write. The conditional function, along with predicates and logical functions, enables students to write programs that perform different actions depending on the nature of the input. Conditionals are represented as one or more test and action boxes, assembled by the student in each problem. A partial solution to a problem from the conditionals curriculum is shown in Figure 9. In this problem, called *successor*, the student has named the two variables *targ* and *lis* to hold the target item and the list being searched, respectively. After the student names a variable, the name appears on the button. The problem also requires the use of two constants, the atoms *no* and *end*, so buttons are provided for these constants. In addition, the widely used constants *t* and *nil* are provided for every problem. Variable buttons are labeled with the word "Var" and the name entered by the student; constant buttons are labeled with the word "Const" and the value of the constant. In the solution shown in Figure 9, the student has used both variables in the first and second test, and has used the constants *end* and *no* in the second and third actions, respectively.

The student in Figure 9 has constructed a solution in the exploratory mode of

Figure 7: Exploring the behavior of a program by running it on new data. After the student types in new input data into the empty input node at the bottom of the screen, the "???" place-holders will be replaced by the new intermediate products and the new final result.
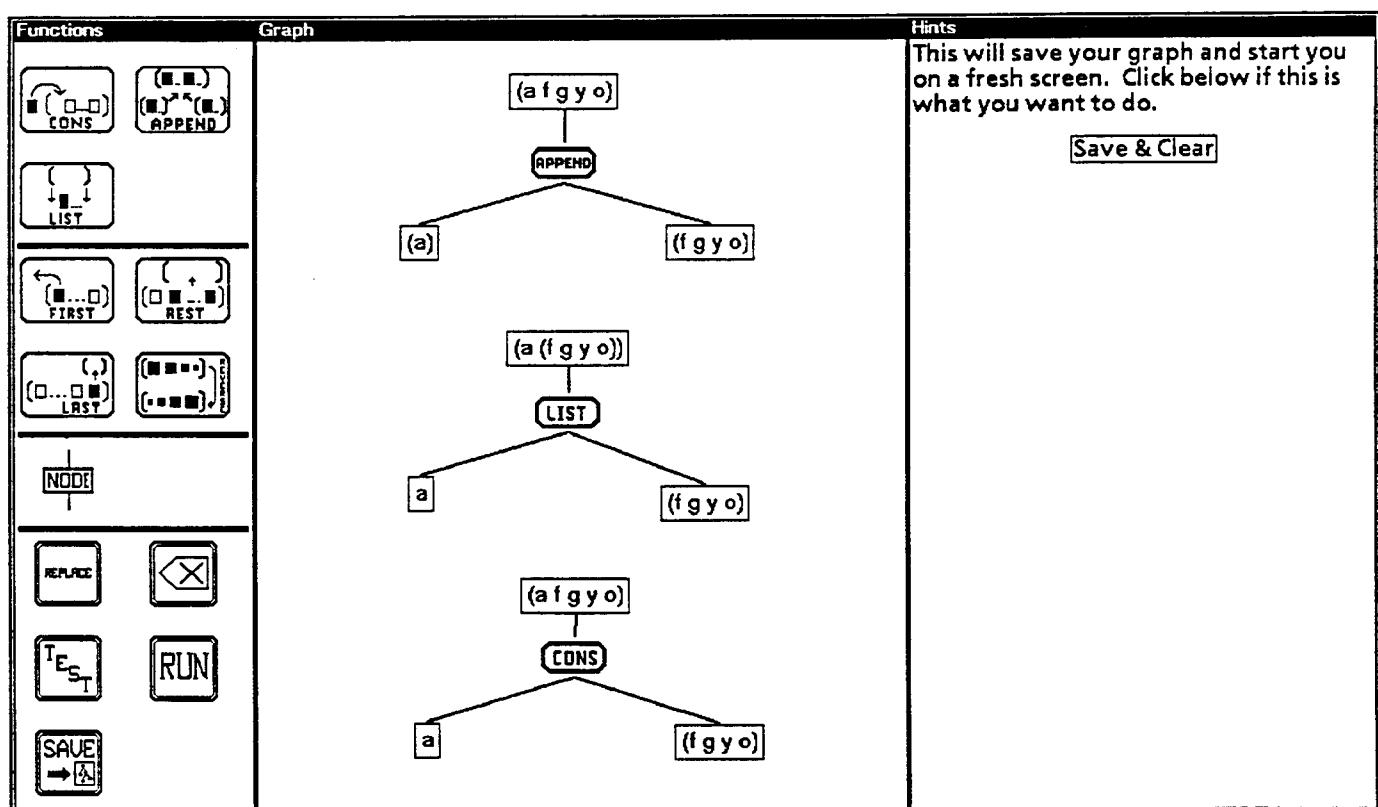
Figure 8: A student saving the results of exploration in GIL.

GIL. When a solution is submitted in this mode, it is evaluated by running the constructed program on a set of test cases. This student's program runs on the tests cases (that is, it does not perform any illegal operations on the data), but it does not produce the correct result.

In some situations GIL suggests sample data to run. In response to the submitted solution in Figure 9, GIL demonstrates the incorrect behavior of the program by running it on one of its test cases. As shown in Figure 10, GIL enters the value *a* for the variable *targ* and *(c b a)* for the variable *lis*. When running a program with variables students can assign particular data to each variable and then observe the propagation of data through the program. The values entered for the variable are shown in each occurrence of the variable box in the program.

When running a program involving conditionals, GIL displays the most important intermediate results (the result of each test and action considered) on the title bar of the corresponding window. The result of the first test, *(a)*, is shown on the title bar of that test box, and the result of the action taken is shown on the title bar of the associated action box (above the test). No other test or action boxes contain results, because the *cond* terminates upon the first successful test. The final result of the program, *nil*, is displayed on its title bar of the *cond*. Thus, in addition to the dynamic displaying of control, students can also see the record of the path through the program. For this example, only the first test has a result, because the second and third tests were never processed. In Figure 11, the first test fails, so the second test is considered. This test succeeds, and so the second action is taken. Hence results are displayed on the title bar for the first and second tests and for the second action.

In addition to observing the intermediate results of each test and action, students can obtain more detailed results by examining the data passed along any connection in the program. Students can click on any darkened link between functions to see the data that passed between them, e.g., the output of *member* that becomes the input of *rest* in the first action in Figure 10. Clicking on the connection between these two functions would cause a box that contains the data *(a)* to "pop up" on top of the line. In this way, this interface conveys the advantages of intermediate products of the simpler list manipulation interface used in earlier chapters, while also providing the opportunity for students to use variables directly in their program, and see the program run with different input values.

Based on understanding why the test case produced the wrong result in Figure 10 the student then modified the program, switching the first and second test-action pair. In addition to re-trying the test data suggested by GIL in Figure 10, the student also tries other runs of the program. One such run is shown in Figure 11. The student then submits the new program, shown in Figure 12, which is now accepted by GIL.

Assignment

Define a function called **successor**. It takes two arguments, a target and a list. If the target is not in the list, the function returns **NO**. If the target is the last item in the list, the function returns **END**. Otherwise, it returns the item that immediately follows the the target in the list.

Hints

Your graph runs.

However you do not have a complete solution for the problem **successor** yet. If you run **successor** with the arguments **a** and **(c b a)**, the final result should be **END**.
Your graph returns **NIL.**

Click on the **Submit** button again, if you want to see it run.

Functions

CONS | APPEND | LIST
FIRST | REST | LAST | REVERSE
LISTP | ATOM | NUMBERP | ZEROP
EQUAL | MEMBER | NULL | NOT
AND | OR | < | >
+ | − | * | /
COND
Var TARG | Var LIS
Const T | Const NIL | Const NO | Const END

Control

Oops | Replace | Delete | Switch | Clear | Run | Step run | Sub -mit

Problem: successor

Cond

Action: FIRST → REST → MEMBER → Var TARG, Var LIS

Action: END

Action: NO

Test: MEMBER → Var TARG, Var LIS

Test: EQUAL → FIRST → REVERSE; Var TARG, Var LIS

Test: T

Figure 9: A partially completed solution to a problem involving conditionals.

Assignment	Hints	
Define a function called **successor**. It takes two arguments, a target and a list. If the target is not in the list, the function returns **NO**. If the target is the last item in the list, the function returns **END**. Otherwise, it returns the item that immediately follows the the target in the list.	Ok, your program runs and produces the result **NIL**. But this is not the right result. **successor** given the the arguments **a** and **(c b a)** should have returned **END** instead of **NIL**. To complete the problem, you have to change your graph to handle this	situation.

Functions

CONS	APPEND	LIST
FIRST	REST	LAST	REVERSE

LISTP	ATOM	NUMBERP	ZEROP
EQUAL	MEMBER	NULL	NOT
AND	OR	<	>
+	−	*	/

COND

Var TARG	Var LIS

Const T	Const NIL	Const NO	Const END

Control

Oops | Replace | Delete | Switch | Clear | Run | Step run | Sub -mit

Problem: successor

Cond ... NIL ...

Action	NIL	Action	Action
FIRST → REST → MEMBER Var TARG: a / Var LIS: (c b a)		END	NO

Test	[a]	Test	Test
MEMBER Var TARG: a / Var LIS: (c b a)		EQUAL FIRST REVERSE Var TARG: a / Var LIS: (c b a)	T

Figure 10: The run of the student's program on test data reveals that the program produces the wrong result.
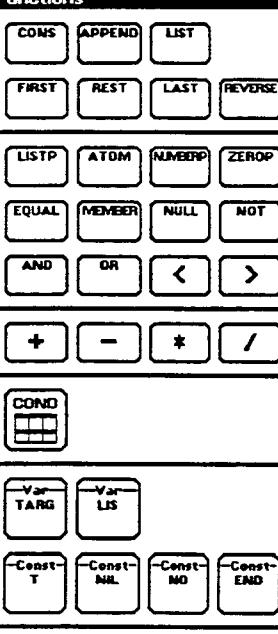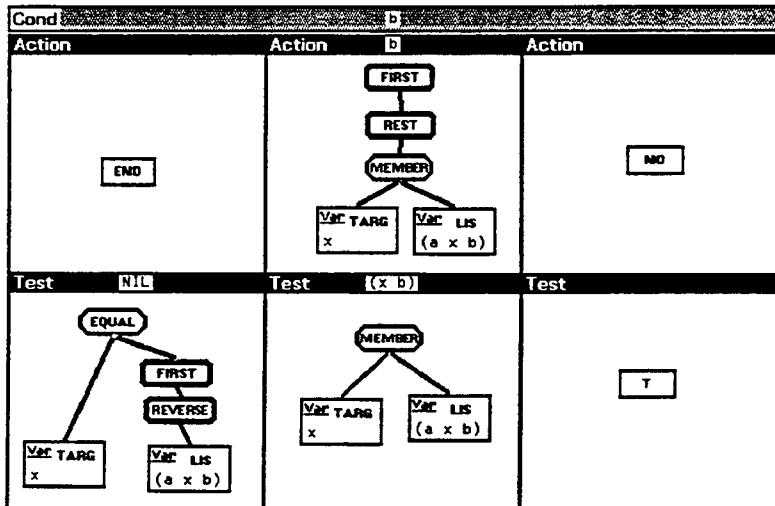
Assignment

Define a function called **successor**. It takes two arguments, a target and a list. If the target is not in the list, the function returns **NO**. If the target is the last item in the list, the function returns **END**. Otherwise, it returns the item that immediately follows the the target in the list.

Hints

This run was succesful.

The value of the **COND** is b.

Functions

| CONS | APPEND | LIST |

| FIRST | REST | LAST | REVERSE |

| LISTP | ATOM | NUMBERP | ZEROP |

| EQUAL | MEMBER | NULL | NOT |

| AND | OR | < | > |

| + | − | * | / |

| COND |

| Var TARG | Var LIS |

| Const T | Const NIL | Const NO | Const END |

Control

| Oops | Replace | Delete | Switch | Clear | Run | Step run | Sub -mit |

Problem: successor

Cond b

Action

END

Action b

FIRST
REST
MEMBER
Var TARG x
Var LIS (a x b)

Action

NO

Test NIL

EQUAL
FIRST
REVERSE
Var TARG x
Var LIS (a x b)

Test (x b)

MEMBER
Var TARG x
Var LIS (a x b)

Test

T

Figure 11: A run of the student's revised program.

Figure 12: The completed solution to this conditionals problem, now accepted by GIL.

Currently, the instructional mode, either model tracing or exploratory, is set by the instructor. In future research, we plan to investigate strategies by which GIL can determine whether it should intervene upon an error or allow the student more freedom to diagnose and repair the error; this decision would be contingent upon the severity of the error and the student's previous successes or difficulties. Dynamically selecting the appropriate pedagogical strategy promises to be an important improvement in the flexibility of interactive learning environments (Lesgold, Lajoie, Logan, & Eggan, 1990; Murray, 1989; Spensley et al., 1990).

In summary, GIL employs a graphical interface and model tracing feedback designed to work together to support students' reasoning. The graphical representation helps students articulate their reasoning and provides a better representation of the structure of the solution. GIL's model tracer structures its feedback around this elaborated visual representation. If GIL is successful, students will learn the constructs and strategies of a programming language more easily using the graphical support, and they will find it easier to recover from obstacles and errors with GIL's explanatory feedback.

3   Formative Empirical Evaluation of GIL

In an initial study, we compared students learning to program using GIL with students working without assistance in a standard LISP programming environment consisting of a textbook, a simple editor, and an interactive LISP interpreter (Reiser et al., 1989). The GIL system appears to be effective in tutoring students through the first few chapters of an introductory LISP curriculum. Students using GIL learned the material more quickly and with less difficulty than students using the standard environment.

If an interactive learning environment does indeed facilitate learning, it is important to determine exactly which aspects of the environment are responsible for the benefits. As described in the previous section, our current research investigates whether graphical interfaces and explanatory feedback do indeed produce pedagogical benefits, using GIL as an experimental testbed. In the following sections we present evidence relevant to each class of benefits and argue that the graphical interface and the explanatory feedback work in concert to produce GIL's pedagogical benefits.

4 GIL Facilitates Reasoning with Congruent Representations

In this section, we consider to what extent the benefits exhibited by GIL students arise from GIL's graphical representation. We expect that students using a visual representation for solutions in complex domains will learn to solve problems more easily than students using traditional representations.

The potential benefits of GIL's graphical representations have several components. The representation renders the behavior of the program more visible and leads students to make their reasoning explicit in the program. It is also designed to match more closely the structure of students' reasoning, and to make it simpler for students to decompose a problem into subgoals and work on those subgoals in whatever order they find natural. These aspects of GIL are designed to work together to minimize the discrepancies between the student's plan and the way in which the solution is communicated. First, we discuss a comparison of students learning with GIL (without its model tracing feedback) with other students learning with traditional text-based representations. Then, we review a study that begins to examine one of the benefits of GIL's representation, the way in which it allows students to reason in a more natural direction.

We have examined a group of students using GIL without its model tracing while working through the first two sections of the curriculum (Reiser, Copen, Ranney, Hamid, & Kimberg, 1991). Recall that when GIL operates without model tracing, it contains the graphical interface and facilities for testing whether a program is correct and for running the program on new input data. In this experiment, students learned sections 1 and 2 in the curriculum (see Table 1). Students work with a concrete example in these problems, specifying intermediate products (as in Figures 2–6).

This exploratory version of GIL was developed primarily to allow a comparison with the full model tracing version of GIL, in order to examine the benefits of model tracing feedback (Reiser, Copen, Ranney, Hamid, & Kimberg, 1991). For our current purpose, however, we can also compare these learning sessions with a control group of students who learned LISP in a standard programming environment containing a simple editor and an interactive LISP interpreter (Merrill et al., 1991). Interestingly, the GIL students solved the problems in the curriculum in approximately half the time required by students using the standard programming environment. Thus, even without model tracing feedback, students using GIL's graphical representation solve problems more easily than students working with the text-based language.

This study showed that GIL offers significant advantages over standard text LISP. The GIL group generally constructed their solutions requiring fewer modifications to their programs, and their modifications were focused almost primarily on semantic and algorithmic changes rather than syntactic changes. In general,

the results suggest that the GIL students were able to construct their solutions more easily, and that they achieved a better understanding of the programs they constructed.

One advantage of GIL's design is that it allows students to work on the components of a solution in whatever order they find most natural. The next experiment examines whether this freedom helps students acquire the target skills with less difficulty. In the early curriculum, when GIL students build a program graph using a particular example and are required to specify all the intermediate products, they are free to reason in whatever direction they find natural. Students can reason either from the inputs toward the goal data (a forward step) or backward from the goal data toward the original inputs. For example, consider again the problem shown in Figure 2. Suppose the first step the student took was to use the function *cons* on inputs *d* and *(a b c)* to obtain *(d a b c)*. This would be a backward step, because the student used a function on new inputs to obtain the goal. Suppose the student then used *last* on *(a b c d)* to produce *(d)*. This function used given data (the original input) to produce new data as output, and thus constitutes a forward step. Students are free in GIL to work on any part of the problem at any point, combining forward and backward reasoning as desired.

Reiser et al. (1989) found that novice programmers strongly preferred to take forward steps in their solutions, suggesting that novices can reason more easily about the behavior of programs by working forward from the input data rather than backward from the goal. Interestingly, the order of reasoning exhibited by GIL subjects is the opposite of the surface form of the solution — the first function encountered in the body of the text form is in fact the last component of the solution assembled by GIL subjects (see Figure 3). If this reliance on forward steps in GIL accurately represents students' reasoning, then one advantage of GIL's graphical system may be that it offers students a way to reason in a more natural order than students who are led to assemble a solution in the order in which the components appear in the text form of the solution (Ranney & Reiser, 1989; Reiser et al., 1989).

We investigated this idea in an experiment which manipulated the type of steps students were allowed to use in constructing solutions (Trafton & Reiser, 1991). Each of three groups of subjects received a different version of GIL. The free subjects were shown both forward and backward steps and were told they could work on any portion of the graph using either type of step. The forward step subjects were required to reason using only forward steps from the input to the goal, while backward step subjects were required to use only steps that chained from the goal toward the original inputs. We found an overwhelming preference for forward working steps in the subjects in the free condition (95% of the steps), replicating our earlier findings about students' judgments of what type of reasoning is more natural (Reiser et al., 1989). These students' preferences appear well founded. Subjects re-

quired to reason using backward steps took approximately 50% longer to solve the assigned problems than subjects who worked using forward reasoning. In addition, the backward working subjects spent more time understanding and correcting errors than the forward working subjects, suggesting that they found it more difficult to reason about their errors. They also spent more time deleting correct portions of their solutions than forward working subjects, suggesting they reached impasses more frequently. These effects are stronger in the lower ability students than the higher ability students.

Students programming in standard text LISP are generally led to enter a LISP function in backward order, like the backward condition of GIL. We postulate that when novice programmers are forced to enter a LISP function in this backward order, they are reasoning with forward steps until a goal is reached and then entering the program with the required backward steps. Forward reasoning may be simpler in this domain because it is more consistent with the naive models students construct about how functions behave and because the search space of options is smaller when reasoning forward (because the output of a function is uniquely determined from its input, but not vice versa). As students become more familiar with the behavior of LISP functions and the particular programming constructs, they begin to learn patterns of code and can work more easily in the backward direction if necessary.

This study suggests that GIL simplifies the task of learning programming by allowing students to work in a direction that is more congruent with the reasoning they do in this domain. The construction of a program graph mirrors the problem solving that students perform to arrive at the final solution. Students working in GIL do not need to reason several steps ahead, as novice students working in text LISP may need to do, which simplifies the planning component of the task. Furthermore, this simplification serves to minimize the amount of information students must keep in memory, helping to reduce the cost of errors (Anderson & Jeffries, 1985) and leaving more memory resources available for acquiring the new skill (Sweller, 1988).

In summary, the graphical representation appears to provide support for students' reasoning. By minimizing the demands of the lower level syntactic details of the programming language, GIL allows students to focus on the more central conceptual issues in the domain. In addition, GIL students easily employ the readily available state information provided by the graphical representation. The closer fit between the structure of a solution in GIL and the students' planning helps them to understand their programs more clearly as they construct them and to focus on the modifications required to repair errors. Finally, GIL enables students to reason in a more natural direction than students using the text form of the language. In general, GIL leads students to consider how their programs behave as they construct them, rather than simply assisting students in assembling correct programs.

5 GIL Facilitates Reasoning with Model Tracing Feedback

Human tutors intervene to help students find their errors and repair them, although in many cases this intervention is very subtle (Fox, 1988; Lepper & Chabay, 1988; Merrill et al., 1991). Furthermore, tutors manage to provide a degree of guidance while leaving much of the control in the hands of their students. Earlier we characterized how GIL can follow students' reasoning and intervene when they make mistakes or pursue a poor strategy. Is this more directive and less subtle form of guidance effective? Are there potential drawbacks to this kind of intervention?

In the previous section, we discussed how GIL supports students' reasoning through its graphical interface. In a series of experiments, we have examined whether the model tracing feedback provided in GIL provides pedagogical benefits beyond those of the graphical interface. In this section, we summarize an experiment that examines the effects of GIL's intervention strategy, as well as a series of experiments examining whether explanatory feedback facilitates reasoning to a greater extent than simply drawing students' attention to errors that have occurred.

We compared students learning to program using the model tracing version of GIL with students using the exploratory version of GIL (without model tracing). This comparison allowed us to examine the costs and benefits of freedom to explore versus the guidance of model tracing feedback (Reiser, Copen, Ranney, Hamid, & Kimberg, 1991). As described earlier, both versions share the same graphical interface, but the exploratory students work without model tracing guidance and have the freedom to make any step, including illegal steps or steps that represent poor strategies. To heighten the contrast, in this study the facilities for testing and running programs were available only to the exploratory subjects, not to the model tracing subjects. Thus, the model tracing students relied on GIL's explanatory guidance, whereas exploratory students received feedback only on request. Furthermore, since the exploratory version is designed to allow students more freedom in solving the problems, it responds only to legal errors when the student requests a test of the graph and does not comment on the student's solution strategy.

The exploratory subjects were able to solve the assigned problems correctly by working on their own, using only the system's testing facilities to get feedback. Approximately one-half of the cases in which a student deleted or replaced a portion of a graph were initiated after the student's test of that graph revealed an error. In these cases, the student built a portion of the program, tested it, found an error, and then tried to repair it. Thus, these students appeared to profitably use the greater freedom provided to manage their own errors. Students constructed a partial solution which they expected to behave in a particular way, tested it and found that it did not behave as predicted, and then began to modify the solution

based upon this information.

Self-initiated modifications of solutions are another important type of problem solving behavior. In one-third of the cases in which the exploratory students deleted or replaced portions of a program, they initiated these modifications without requesting any feedback. These episodes suggest that students are often able to find their own errors and repair them, an activity some theorists argue is central to learning by doing (Collins & Brown, 1988; Schank & Farrell, 1987; Schank & Jona, 1991; Schwartz, 1989).

The learning outcomes of the two groups were not identical. Not surprisingly, the subjects left free to explore took almost twice as long to solve the assigned problems as the model tracing subjects, who received feedback as soon as they committed errors. Both groups learned to construct programs approximately equally well, as evidenced by their performance on a posttest. However, the groups differed in their abilities to debug programs containing errors. The exploration subjects were better able to find errors in buggy programs than the model tracing subjects. This suggests that the exploratory subjects' more active role in finding and fixing their own bugs, although leading to longer learning times, paid off in superior error recovery skills.

The groups also differed in their motivational outcomes, but the nature of the motivational consequences appeared to depend upon the relative ability of the student. High ability students exhibited more positive opinions of their abilities and of programming when given the freedom to explore rather than the more constraining model tracing environment. The pattern was reversed for lower ability students. Lower ability students, who encountered substantially more errors and required much more time in the exploratory condition, exhibited more negative judgments about the domain and held lower opinions of their abilities than comparable students who received more active guidance from the model tracing system. These results are consistent with Snow and Lohman's (1984) review of learning outcomes, in which they suggest that structure and guidance are more important for low ability learners. Although the interaction of motivational outcome with ability in our study is based on relatively small samples at each ability level, the results do suggest that there is no uniform motivational cost or benefit of either pedagogical style. Instead, the greater freedom of the exploratory system yields benefits for students who can capitalize on that freedom, but may lead to negative consequences for students who have more difficulty solving the problems without a tutor's guidance. Interestingly, the model tracing group results also suggest that tutoring can overcome negative attitudes often found in lower ability students, who rated their enjoyment of the learning and their own abilities as highly as the high ability students did. These results support the claims made by Lepper and his colleagues (Lepper et al., 1990; Lepper & Chabay, 1988) about the potential for positive motivational outcomes of tutoring for students who encounter difficulty in a domain.

GIL's model tracing feedback does help students solve the problems more easily, and for some students this appears to lead to positive motivational outcomes. The next issue to examine is the content of the feedback. GIL intervenes upon errors, offering an explanation of what is wrong with the student's step or how it might be improved. Is the intervention itself sufficient to focus students' attention on the occurrence and location of errors, or do students indeed rely on the content of the explanations to understand their errors and repair them? Human tutors often simply focus students' attention on problematic portions of a solution, helping them realize that something is wrong, rather than providing detailed explanations of why the solution is incorrect (Merrill et al., 1991). Do students successfully use the more direct explanatory feedback offered by intelligent tutoring systems to learn from their errors?

To investigate this issue, we conducted a series of experiments that examine how different content in GIL's feedback can affect students' learning (Reiser, Connelly, Ranney, & Ritter, in preparation). We constructed two variants of GIL that provided different types of feedback upon students' errors. The minimal feedback version provided information only regarding whether the step was correct or not; it also provided students the opportunity to request the correct step, without an explanation of either the error or the correct step. The location feedback version provided information about which part of the step was incorrect; again, it indicated how to fix the step upon request, but without any explanation of the repair. The third condition was GIL with its explanatory feedback. To ensure that students relied on the explanatory feedback, we did not provide any testing facilities in these experiments. In a first experiment, subjects in the minimal and location feedback conditions relied more on the second level of help, more frequently "giving up" and asking for the correct step. These subjects also exhibited longer error-fixing episodes and poorer performance on posttests than subjects receiving explanatory feedback upon errors. In two additional experiments comparing minimal and explanatory feedback groups, the "More Info" option was removed, to investigate whether the differences in learning session time and learning outcomes were simply due to the passive learning strategy used by the minimal feedback subjects. Not surprisingly, removing the option of being told the answer resulted in generally longer learning times and more errors. However, in these experiments, subjects in the minimal feedback condition also showed poorer performance during the learning sessions and on posttests than subjects receiving explanatory feedback, replicating the results of the earlier experiment. The minimal feedback subjects made many more errors, deleted more of their correct steps, took longer to solve the assigned problems, and performed more poorly on some of the posttests.

These results suggest that explanatory feedback provided in a model tracing tutor can speed learning. It reduces error recovery attempts, reduces overall learning

time, and reduces the frequency of deleting partial solutions. Students who receive only minimal feedback upon errors appear to encounter more difficulties and master the material less well than students who are given hints and can construct their own explanations. Furthermore, if students can ask for the right answer, then reduced feedback may lead to a strategy of relying too heavily on the feedback rather than engaging in the reasoning to construct answers themselves, which is generally viewed as a less effective learning strategy (e.g., Bangert-Drowns, Kulik, Kulik, & Morgan, 1991; Kulhavy, 1977).

In summary, these studies suggest that GIL's facilitation of students' learning is not merely due to providing feedback on the correctness of the steps nor in leading students to a solution by telling them the answer. Rather, the positive effects of GIL lie in the way the explanations enable students to correct their errors and understand why their repairs are successful. These results suggest that explanatory feedback can help students understand how their programs actually work as they learn to construct programs, rather than merely learning correct sequences of programming constructs that achieve particular purposes.

6   Conclusions

We have seen that GIL can achieve some of the success of human tutors, even though its behavior is somewhat less subtle. A computer tutor built according to principles like those embodied in GIL can assist students in several key aspects of learning by doing. Students are left free to handle as much of the problem solving effort as they can. When necessary, the tutor can provide guidance to help prevent floundering and help students detect, understand, and repair errors. Thus, the tutor ensures that the problem solving stays on a productive track. Such a tutor can assist students in solving problems that might otherwise be beyond their competence, providing help only when needed, so that students can eventually master the skills and solve the problems on their own.

Our results suggest that GIL facilitates learning by helping students build a more complete understanding of the domain as they solve problems. GIL achieves this through its explanatory feedback and graphical representations. The explanatory feedback assists students in building an elaborated model — students are not only able to assemble components together to achieve a task, but they are encouraged to explain *why* that collection of components behaves as it does. An important aspect of GIL's support in this learning lies in its graphical representations. The graphical representation requires students to make explicit predictions about the internal states of the devices they are constructing and provides a better match with students' reasoning than the more cumbersome text representation.

These results also have implications for the nature of effective problem solv-

ing knowledge. The results suggest the importance of an underlying understanding or mental model in addition to the acquisition of rule and plan-based procedures. Much research in machine and human learning has focused on the acquisition and modification of procedures (Anderson, 1983; Newell, 1990). Although expert knowledge may ultimately consist primarily of well learned compiled procedures, our work emphasizes the importance of supporting novices' acquisition of explanatory models that can be used to reason prior to well-learned procedures. Novices must be able to reason about possible causes and effects of an action in order to select the next step in a solution. GIL's graphical representation helps scaffold reasoning within such a model, and its causal explanations help guide students as they attempt to employ the model to make predictions.

We believe that the tutoring methodologies employed in our GIL research would be useful for domains other than computer programming. The characteristics of the domain important for the benefits of GIL's design are that students have to build objects or analyze their behavior. A generalization of the techniques clearly offers promise for domains in which students must analyze devices, such as electronics or mechanical troubleshooting. Furthermore, many scientific domains may be modeled as the behavior of causally interacting objects. Our research suggests the potential benefits of employing causal explanations and graphical representations to help students understand the behavior of objects in the target domain. Graphical representations might be employed to provide support for students as they construct a model of the target system. If students could be led to make predictions about the states at various points, the system could then offer causal explanations based on these representations. In general, the results indicate the exciting potentials for a computerized tutor to support and guide students' reasoning in these types of problem solving domains.

References

Anderson, J. R. (1983). *The architecture of cognition.* Cambridge, MA: Harvard University Press.

Anderson, J. R., Boyle, C. F., Corbett, A. T., & Lewis, M. W. (1990). Cognitive modeling and intelligent tutoring. *Artificial Intelligence, 42,* 7–49.

Anderson, J. R., Boyle, C. F., Farrell, R. G., & Reiser, B. J. (1987a). Cognitive principles in the design of computer tutors. In P. Morris (Ed.), *Modelling cognition.* New York: John Wiley and Sons.

Anderson, J. R., Boyle, C. F., & Reiser, B. J. (1985). Intelligent tutoring systems. *Science, 228,* 456–462.

Anderson, J. R., Boyle, C. F., & Yost, G. (1986). The geometry tutor. *Journal of Mathematical Behavior, 5,* 5–19.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science, 13,* 467–505.

Anderson, J. R., Corbett, A. T., & Reiser, B. J. (1987b). *Essential LISP.* Reading, MA: Addison-Wesley.

Anderson, J. R. & Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human-Computer Interaction, 1,* 107–131.

Anderson, J. R. & Reiser, B. J. (1985). The LISP tutor. *Byte, 10,* 159–175.

Bangert-Drowns, R. L., Kulik, C.-L. C., Kulik, J. A., & Morgan, M. T. (1991). The instructional effect of feedback in test-like events. *Review of Educational Research, 61,* 213–238.

Bayman, P. & Mayer, R. E. (1988). Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology, 80,* 291–298.

Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher, 13,* 4–16.

Brown, J. S. (1985). Process versus product: A perspective on tools for communal and informal electronic learning. *Journal of Educational Computing Research, 1,* 179–202.

Carbonell, J. R. (1970). AI in CAI: An articificial intelliegence approach to computer-aided instruction. *IEEE Transactions on Man-Machine Systems*, 11, 190–202.

Clancey, W. J. (1987). *Knowledge-based tutoring: The Guidon program.* Cambridge, MA: MIT Press.

Cohen, P. A., Kulik, J. A., & Kulik, C.-L. C. (1982). Educational outcomes of tutoring: A meta-analysis of findings. *American Educational Research Journal*, 19, 237–248.

Collins, A. & Brown, J. S. (1988). The computer as a tool for learning through reflection. In H. Mandl & A. Lesgold (Eds.), *Learning issues for intelligent tutoring systems.* New York: Springer-Verlag.

Collins, A., Brown, J. S., & Newman, S. E. (1989). Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. In L. B. Resnick (Ed.), *Knowing, learning, and instruction: Essays in honor of Robert Glaser* (pp. 453–494). Hillsdale, NJ: Erlbaum.

Collins, A. & Stevens, A. L. (1982). Goals and strategies of inquiry teachers. In R. Glaser (Ed.), *Advances in instructional psychology, Volume 2.* Hillsdale, NJ: Erlbaum.

Collins, A., Warnock, E. H., & Passafiume, J. J. (1975). Analysis and synthesis of tutorial dialogues. In G. H. Bower (Ed.), *The psychology of learning and motivation* (pp. 49–87). New York: Academic Press.

de Kleer, J. & Brown, J. S. (1983). Assumptions and ambiguities in mechanistic mental models. In D. Gentner & A. L. Stevens (Eds.), *Mental models.* Hillsdale, NJ: Erlbaum.

du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14, 237–249.

Fox, B. A. (1988). *Cognitive and interactional aspects of correction in tutoring.* Technical Report No. 88-2, Institute of Cognitive Science, University of Colorado, Boulder, Colorado.

Fuson, K. C. & Willis, G. B. (1989). Second graders' use of schematic drawings in solving addition and subtraction word problems. *Journal of Educational Psychology*, 81, 514–520.

Genesereth, M. R. (1982). The role of plans in intelligent teaching systems. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 137–155). London, UK: Academic Press.

Kimball, R. (1982). A self-improving tutor for symbolic integration. In D. H. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 283–307). London: Academic Press.

Kulhavy, R. W. (1977). Feedback in written instruction. *Review of Educational Research*, 47, 211–232.

Larkin, J. H. (1989). Display-based problem solving. In D. Klahr & K. Kotovsky (Eds.), *Complex information processing: The impact of Herbert A. Simon*. Hillsdale, NJ: Erlbaum.

Larkin, J. H. & Simon, H. A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11, 65–99.

Lepper, M. R., Aspinwall, L., Mumme, D., & Chabay, R. W. (1990). Self-perception and social perception processes in tutoring: Subtle social control strategies of expert tutors. In J. M. Olson & M. P. Zanna (Eds.), *Self inference processes: The sixth Ontario symposium in social psychology*. Hillsdale, NJ: Erlbaum.

Lepper, M. R. & Chabay, R. W. (1988). Socializing the intelligent tutor: Bringing empathy to computer tutors. In H. Mandl & A. Lesgold (Eds.), *Learning issues for intelligent tutoring systems* (pp. 242–257). New York: Springer-Verlag.

Lesgold, A., Lajoie, S., Logan, D., & Eggan, G. (1990). Applying cognitive task analysis and research methods to assessment. In N. Frederiksen, R. Glaser, A. Lesgold, & M. G. Shafto (Eds.), *Diagnostic monitoring of skill and knowledge acquisition* (pp. 325–350). Hillsdale, NJ: Erlbaum.

Merrill, D. C., Reiser, B. J., Ranney, M., & Trafton, J. G. (1991). Effective pedagogical techniques in human tutors and intelligent tutoring systems. Manuscript submitted for publication.

Milson, R., Lewis, M. W., & Anderson, J. R. (1990). The teacher's apprentice: Building an algebra tutor. In R. Freedle (Ed.), *Artificial intelligence and the future of testing*. Hillsdale, NJ: Erlbaum.

Murray, W. R. (1989). Control for intelligent tutoring systems: A blackboard-based dynamic instructional planner. In D. Bierman, J. Breuker, & J. Sandberg (Eds.), *Proceedings of the Fourth International Conference on Artificial Intelligence and Education* (pp. 150–168). Springfield, VA: IOS.

Newell, A. (1990). *Unified theories of cognition.* Cambridge, MA: Harvard University Press.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas.* New York: Basic Books, Inc.

Ranney, M. & Reiser, B. J. (1989). Reasoning and explanation in an intelligent tutoring system for programming. In G. Salvendy & M. J. Smith (Eds.), *Designing and using human-computer interfaces and knowledge based systems: Proceedings of the Third International Conference on Human-Computer Interaction* (pp. 88–95). New York: Elsevier Science Publishers.

Reiser, B. J., Anderson, J. R., & Farrell, R. G. (1985). Dynamic student modeling in an intelligent tutor for LISP programming. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* Los Angeles, CA.

Reiser, B. J., Connelly, J. W., Ranney, M., & Ritter, C. (in preparation). The role of explanatory feedback in skill acquisition. Manuscript in preparation, Princeton University.

Reiser, B. J., Copen, W. A., Ranney, M., Hamid, A., & Kimberg, D. Y. (1991a). Cognitive and motivational consequences of tutoring and discovery learning. Manuscript, Princeton University.

Reiser, B. J., Friedmann, P., Gevins, J., Kimberg, D. Y., Ranney, M., & Romero, A. (1988a). A graphical programming language interface for an intelligent LISP tutor. In *Proceedings of CHI'88 Conference on Human Factors in Computing Systems* (pp. 39–44). New York.

Reiser, B. J., Friedmann, P., Kimberg, D. Y., & Ranney, M. (1988b). Constructing explanations from problem solving rules to guide the planning of programs. In *Proceedings of ITS-88: The International Conference on Intelligent Tutoring Systems* (pp. 222–229). Montreal.

Reiser, B. J., Kimberg, D. Y., Lovett, M. C., & Ranney, M. (1991b). Knowledge representation and explanation in GIL, an intelligent tutor for programming. In J. H. Larkin & R. W. Chabay (Eds.), *Computer assisted instruction and intelligent tutoring systems: Shared issues and complementary approaches.* Hillsdale, NJ: Erlbaum.

Reiser, B. J., Ranney, M., Lovett, M. C., & Kimberg, D. Y. (1989). Facilitating students' reasoning with causal explanations and visual representations. In

D. Bierman, J. Breuker, & J. Sandberg (Eds.), *Proceedings of the Fourth International Conference on Artificial Intelligence and Education* (pp. 228–235). Springfield, VA: IOS.

Schank, R. C. & Farrell, R. (1987). Creativity in education: A standard for computer-based teaching. *Machine-Mediated Learning*, 2, 175–194.

Schank, R. C. & Jona, M. Y. (1991). Empowering the student: New perspectives on the design of teaching systems. *The Journal of the Learning Sciences*, 1, 7–35.

Schwartz, J. L. (1989). Intellectual mirrors: A step in the direction of making schools knowledge places. *Harvard Educational Review*, 59, 51–61.

Singley, M. K. (1990). The reification of goal structures in a calculus tutor: Effects on problem solving performance. *Interactive Learning Environments*, 1, 102–123.

Snow, R. E. & Lohman, D. F. (1984). Toward a theory of cognitive aptitude for learning from instruction. *Journal of Educational Psychology*, 76, 347–376.

Spensley, F., Elsom-Cook, M., Byerley, P., Brooks, P., Federici, M., & Scaroni, C. (1990). Using multiple teaching strategies in an ITS. In C. Frasson & G. Gauthier (Eds.), *Intelligent tutoring systems: At the crossroads of artificial intelligence and education*. Norwood, NJ: Ablex Publishing Corporation.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12, 257–285.

Trafton, J. G. & Reiser, B. J. (1991). Providing natural representations to facilitate novices' understanding in a new domain: Forward and backward reasoning in programming. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society* (pp. 923–927). Chicago, IL.

White, B. Y. & Frederiksen, J. R. (1990). Causal model progressions as a foundation for intelligent learning environments. *Artificial Intelligence*, 42, 99–157.

Young, R. M. (1981). The machine inside the machine: users' models of pocket calculators. *International Journal of Man-Machine Studies*, 15, 51–85.